The Intersoft C compiler

Version 2.5

TRS80 Model I Implementation Manual

For TRS80 Model I micro-computers with
TRSDOS (except version 1.3) or LDOS

This document is subject to change without notice and does not represent a commitment on the part of Intersoft Unlimited. All Intersoft Unlimited programs are distributed on an "as is" basis without warranty. Intersoft Unlimited shall have no responsiblity to any customer, person or entity with respect to any liability, loss or damage caused by programs sold by Intersoft Unlimited.

Intersoft is not a large company, we are a group of professional programmers endeavouring to provide high quality software for as low a price as we can manage. There is little that we can do to prevent software piracy other than to request that you, our fellow programmers, refrain from this common practice.

Please report any inaccuracies in the documentation by completing and returning the "Reader Comments" form included at the end of the Programmer's Manual.

C O N T E N T S

DISK CONTENTS

Disk #1:

C/CMD	The C compiler
CLIB/REL	Linker library of run time support functions
STDIO/H	The standard I/O constants file
RUNLIB/MAC	The 16-bit run time support
CPY/C	Source for the copy program
README/TXT	Final notes

Disk #2:

IO/H	Constants and globals for "IO/C"
OS/H	Constants and globals for "OS/C"
PRINTF/H	Constants and globals for "PRINTF/C"
SCANF/H	Constants and globals for "SCANF/C"
IO/C	I/O support source library
OS/C	OS support source library
PRINTF/C	printf(), fprintf() and sprintf() source library
SCANF/C	scanf(), fscanf() and sscanf() source library
STD1/C	Miscellaneous support source library 1
STD2/C	Miscellaneous support source library 2
STD3/C	Miscellaneous support source library 3
PROLOG/MAC	Assembler prologue for all C programs
EPILOG/MAC	Routines to call the Operating System
MERGE/C	Source for the merge program
SPLIT/C	Source for the split program
EXTRACT/C	Source for extract program

The compiler source (if ordered) resides on disks #3 and #4.

CHECKOUT PROCEDURE

1) If you are wise you will back up the distribution disks immediately.

2) - Make a system disk for C compilations containing the following files from the distribution disks:

     C/CMD
     STDIO/H
     CPY/C

     We recommend that you kill all non-essential files from this system disk and attempt to add your editor to the disk. For information on what files you don't need on your system disk consult your DOS manual. Eg. For TRSDOS 2.3 the files BASIC/CMD, BASICR/CMD and FORMAT/CMD may be removed; their passwords are BASIC, BASIC and FORMAT respectively.

   - Make a second system disk containing the Microsoft M80 assembler, L80 linker and the file CLIB/REL from the distribution disks.

   - Make a data diskette to containing the file CPY/C from the distribution disks.

3) With the first system disk prepared in step 2) in drive 0: and the data diskette in drive 1: type the following command:

   ```
   C CPY/C:1 O=CPY/MAC:1
   ```

   - At this point load the second system disk prepared in step 2) into drive 0:, then type:

   ```
   M80 CPY:1=CPY:1
   KILL CPY/MAC:1
   L80 CPY:1,CLIB-S,-U,CPY:1-N-E:CCMAIN
   KILL CPY/REL:1
   ```

   This illustrates the normal sequence of compiling, assembling and linking a C program. Compiler options are described in the section of the Programmer's manual titled "How to use the Compiler".
   The library module "CLIB" must be included in the link line or there will be unresolved external references.
   All Intersoft C programs begin execution at the label "CCMAIN".

   If the compiler, assembler or linker report errors then

retry the process recording what you typed and what errors were reported before contacting the retailer.

4) Type the command:

CPY:1

Now type some lines. You will find that the TRS80 line editing commands will work. All the program does in this form is echo what was typed.

The full abilities of this program are explained in the section of the Programmer's manual titled "Sample C Programs". To terminate the program press BREAK to enter an End of File code for *BI. It is possible that the BREAK key will not enter an End of File under some versions of TRSDOS. If the BREAK key does not terminate the CPY program under your DOS please refer to note 14) in the "Special Notes" section of this manual.

The compiler is undamaged if you are able to complete these four steps. The remainder of the files on the distribution disks are not used during normal C compilations. The source code for all of the run time support functions (and the compiler source if it was ordered) have been provided to allow you to customize the compiler.

SPECIAL NOTES FOR THE TRS80 MODEL I IMPLEMENTATION

1) Only the first six characters of global symbol names are significant with the MACRO80 assembler.

2) The TRS80 version of the compiler uses the restart vectors 3 and 4. If these restart vectors are unavailable on your machine then the compiler will not work correctly.

3) There are no default extensions for either the input or output files to the compiler.

4) Do not specify the same input and output file name to the compiler. The file will be overwritten by the output of the compiler faster than the file is read for input to the compiler. This will eventually cause the compiler to attempt to compile its own output.

5) The function call "exit(n)" will abort a "Chain" or "DO" file in progress if "n" is non-zero.

6) The buffered keyboard device (*BI) cannot be re-opened when it is already open.

7) The function call "feof(fp)" checks for End of File by examining the file error code (returned by ferror(fp)) to see if it is 0x1C00 (End of File) or 0x1D00 (No Record Found). When reconfiguring the compiler for a different DOS be certain to verify that these constants are adjusted to reflect the new DOS' conventions.

8) All internal labels used by the compiler and run time support functions are prefixed with "cc". Function and variable names should not begin with "cc" or name clashes might result.

9) The maximum number of simultaneously open files is 14.

10) The special functions "read" and "write" supplied with the TRS80 version do not have the standard arguments as outlined in the book by Kernighan and Ritchie. Both these functions require that the file be opened and closed via fopen() and fclose() rather than open() and close(). The reason for this is that Intersoft C does not yet have the open() and close() functions.

11) There are two significant differences between the TRS80 Model I and Model III. If you attempt to move the compiler to a Model III system then the address of the top of memory pointer must be changed in IO/H and the memini() function of OS/C. The address is 0x4049 on the Model I. The address of the DOS command line should be changed in IO/H if necessary. The address is 0x4318 on the Model I.

12) If your application requires extremely fast i/o you can make use of the following (non-standard) functions from IO/C:

codvr, prdvr, keydvr, bufdvr, ccoutput, ccinput, ccatput, ccatget, ccfspec, ccpeof, ccinit, ccopen and ccclos.

None of these functions are portable to other implementations of C. All of these functions should be used with extreme caution.

13) The source for all the functions in the linker library has been provided with the compiler. Frequently, groups of functions have been collected into a simple form of source library. Whenever functions have been collected into a source library they have been put into the source library in the order in which they were put into the linker library. The ordering of source files and libraries as they went into the linker library is:

prolog/mac, scanf/c, printf/c, std1/c, io/c, std2/c, os/c, std3/c, runlib/mac and epilog/mac.

14) The End of File code for the device *BI (buffered keyboard input) is normally the BREAK key. It is possible that your DOS has disabled the BREAK key. If this is so we recommend that you code programs so that they use some other method of terminating other than checking for End of File from the *BI device. Eg: use the strings "stop" or "end" to indicate that the program should be terminated.

15) The End of File code for the device *CI (unbuffered keyboard input) is normally the control-z key. Under LDOS this corresponds to pressing the shift, down-arrow and Z keys simultaneously. Under TRSDOS this usually corresponds to pressing the shift and down-arrow keys simultaneously.

I/O DEVICES

Intersoft C supports the following devices, they may be used as file names to the "fopen" function:

Name	Description
*BI	Buffered console input.
*CI	Unbuffered console input.
*CO	Unbuffered console output.
*PR	Unbuffered printer output.

The "fopen" function will only allow a device to be opened for its intended purpose. Eg: *BI must be opened with a mode of "r". The output devices may be opened for either write ("w") or append ("a"). The device names may be given in upper or lower case, "fopen" is insensitive to the case of its arguments. To enter an end of file code for *BI press BREAK. To enter an end of file code for *CI press BREAK or control-Z. It is possible that your DOS disables the BREAK key. If so then you will find this out when you perform the acceptance procedure outlined in this manual.

INTERFACING TO NON-C CODE

The compiler generates code in highly predictable patterns, which permits simple interfacing to code written in other languages. The preferred method of interfacing to "foreign" code is by means of function calls.

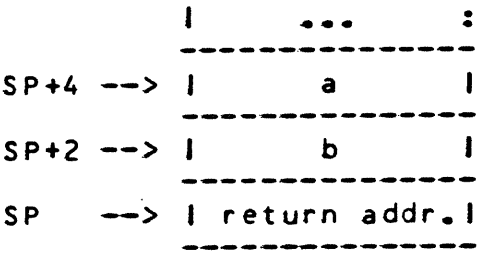
C code can call a function without knowing the nature of the code it contains: there is no checking for argument compatibility. The only restriction is that argument and return value passing must be compatible.

C evaluates function arguments from left to right, and pushes the values on the stack as they occur. Next the number of arguments is loaded into register A (unless the "-n" option is used during compilation) and the function is called. Every argument is stored as two bytes regardless of its actual size.

Calling

```
somefunc(a,b);
```

results in the following machine state on entry to the function:



Register A contains the value 2 (unless the "-n" option was used during compilation).

In the above example, if "somefunc" were an assembler routine, the values of the parameters could be found by applying an offset to the stack pointer. For example, we could fetch the value of b by one of the following code sequences.

```
; fetch the address of the variable
     LD   HL,2          ;offset in bytes from the CURRENT
                        ;stack pointer
     ADD  HL,SP         ;actual address into HL
;
; now fetch the integer value into HL
;
     LD   A,(HL)
     INC  HL
     LD   H,(HL)
     LD   L,A
```

or

```
LD   HL,4        ;the variable's stack offset + 2
CALL CCGIS       ;one of the compiler's 16-bit support
                 ;routines.
```

After any C function call code is generated to reclaim the stack space used by the parameters.

The result of the function call (the returned value) is assumed to be in the register pair HL.

The #asm Preprocessor Directive

This preprocessor directive may be used to insert in-line assembly code within a C function. Although this does not provide as "clean" an interface as calling assembler functions, it is useful at times.

The "#asm" directive causes the source file to be copied (without processing) directly to the output file up to the occurance of the first "#endasm" preprocessor directive. The compiler will not recognise any data definitions, function declarations or preprocessor directives (other than "#endasm") within a "#asm" block. Please note that the text within a "#asm" block is not preprocessed, therefore macros (defined via "#define") will not be expanded. Further restrictions are:

- A "#asm" block may not occur inside a simple statement or expression.

- The stack pointer must be the same on exit as it was upon entry to the "#asm" block.

In order to access variables, including stack variables and function parameters, it is possible to make use of the fact that the code generator leaves the result of the last expression in the register pair HL. The following example doubles the value of "i" and stores it in "j":

```
func() {
  int i, j;

  i;     /* Gets "i" into the register pair HL */
#asm
  ADD HL,HL  ;Double the register pair HL.
  EX  DE,HL  ;Save the result in the register pair DE.
#endasm
  &j;    /* Gets the address of "j" into HL */
#asm
  LD  (HL),E  ;Store the low byte.
  INC HL
  LD  (HL),D  ;Store the high byte.
#endasm
```

A valid use for a "#asm" block within a function containing C code is to access the number of arguments passed to a function. Examples of this are the listings of the functions printf, fprintf and sprintf in the "Sample C programs" section of the Programmer's manual.

SPECIAL FUNCTIONS

In addition to the functions listed in the section of the Programmer's manual titled "Run Time Support Functions", your run time support library contains some functions specifically oriented to TRS80 Model 1 machines.

If you wish to change any of the run time support functions we strongly suggest that you make a library or module of your changed functions and link it before searching the supplied library, rather than actually changing the supplied library.

The following is a list of the special run time support functions (in alphabetical order):

clearerr(fp) - Remove an error condition on a file.

This function returns EOF if the file pointer (fp) is invalid. Otherwise zero is returned and the error flag (returned by ferror(fp)) is cleared to allow further attempts at I/O. No attempt is made to remove the source of the error.

ferror(fp) - Return the last error on a file.

This function returns EOF if the file pointer (fp) is invalid. Otherwise the DOS error code left shifted eight bits is returned. The DOS error codes are described in the technical section of your DOS manual.

read(fp, buf, n) - Read bytes.

This function attempts to read "n" bytes from the file denoted by the file pointer "fp" into the buffer at address "buf". The function returns the number of bytes actually transferred to the buffer. If no bytes were transferred then 0 is returned to indicate end of file or -1 is returned to indicate an error. When this function returns a number different than "n" the next read on the file will return either 0 or -1 to indicate the reason why no more data has been read.

This is a binary read, there are no translations performed on the input stream (ie. carriage-return is not translated to '\n').

See also: write(fp, buf, n), feof(fp), ferror(fp) and clearerr(fp).

setstack() - Set the maximum stack size required.

The default stack size for Intersoft C programs is 1024 bytes. If more is needed then write the function:

```
/* The stack size desired */
#define STACKSZ 2048

int ccstkmg;
setstack()
  ccstkmg = STACKSZ;
```

compile it with the "-g" option, and link it with your program. The free memory pool resides between the end of the program in memory and the top of the stack. Failure to set the stack size appropriately will result in dynamically allocated storage being overwritten. Please note that the I/O support functions use dynamic memory allocation when opening files (even the standard input, output and error files).

sysmsg(ptr) - Write a message to the console.

This function calls the DOS to write the NUL (zero) terminated string at address "ptr" directly to the system console. It is typically used for special errors which prevent the use of the standard error file (stderr) (ie. Reporting an i/o error on the standard error file).

write(fp, buf, n) -Write bytes

This function attempts to write "n" bytes to the file denoted by the file pointer "fp" from the buffer at address "buf". The function returns the number of bytes actually written to the file. A returned value other than "n" indicates an error of some sort. This is a binary write, there are no translations performed on the output stream (ie. '\n' is NOT translated to a carriage-return).

See also: read(fp, buf, n), feof(fp), ferror(fp) and clearerr(fp).

EXTRACT

The "extract" program allows the operator to extract one or more modules from one or more source librarys. This module would then be customized, compiled, assembled and added to a new library or linked with a program before clib/rel. To use the extract program simply type its name. The program will prompt you for the source library name(s), module name(s) and output file name(s). Entering a null line in response to the "Module name ?" query will cause the program to stop extracting modules and ask for a new source library. Entering a null line in response to the "Library name ?" query will cause the program to terminate.

SPLIT AND MERGE

The "split" and "merge" programs are used when a source library is to be compiled and assembled as separate modules under the control of one "DO" file. The "split" program makes individual files for each module in the source library, the "merge" program will create a "DO" file to process the modules. The average Intersoft C user will never need to use these programs, they have been included on the distribution disks because they were used as examples of C programs in the Programmer's manual.

The "split" program splits a source library into its component modules, producing a file containing the names of the modules in the source library. To use this program type:

```
split <library >names
```

where "library" is the name of the source library to be split and "names" is the name of the file to contain the names of the modules. The program also produces one output file per module in the source library (these files are created on drive :0 if they do not already exist). These names of these output files are the names of the modules in the library.

Next the operator is expected to make a template of the "DO" file to process one module, except put a percent (%) character everywhere that the module name would normally be. An example is:

```
C % o=%/MAC -G -N F=1
KILL %
M80 %=%
KILL %/MAC
```

the "merge" program uses this template file and the list of names of modules produced by the "split" program to produce a larger "DO" file which will process each of the modules in turn.

If the output of the split program was:

```
MOD1
MOD2
```

then the output of the "merge" program (with the sample template) would be:

```
C MOD1 o=MOD1/MAC -G -N F=1
KILL MOD1
M80 MOD1=MOD1
KILL MOD1/MAC
C MOD2 o=MOD2/MAC -G -N F=1
KILL MOD2
M80 MOD2=MOD2
KILL MOD2/MAC
```

To use the "merge" program type:

```
merge template <names >output
```

where "template" is the name of the template file, "names" is the name of the file containing the module names and "output" is the name of the output "DO" file.

The Intersoft C compiler

Version 2.5

Programmer's Manual and Reference Guide

For the TRS80, CP/M and FLEX micro-computers

This document is subject to change without notice and does not represent a commitment on the part of Intersoft Unlimited. Reproduction of this manual in whole or in part by any means is forbidden without prior permission from Intersoft Unlimited.

Please report any inaccuracies in the documentation by completing and returning the Reader Comments form included at the end of this manual.

C O N T E N T S

INTRODUCTION

"C" is a general purpose programming language suitable for solving a large class of programming problems. It is not a particularly "high" level language, nor is it particularly "big" in that it does not support a great many control structures. C is a language carefully designed to be a good balance between the concepts that programmers think in terms of and the current capabilities of computers. The fact that computers are becoming faster and more powerful has aided the spread of the C language throughout most classes of computers.

C was originally designed by Dennis Ritchie for the UNIX operating system to run on a DEC PDP-11. Since that time the C language has migrated to mainframe computers (Honeywell 6000 series and IBM 370), to other minicomputers (Interdata 8/32) and to microcomputers (CP/M, FLEX and TRS80).

An excellent reference to the C programming language is "The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie published by Prentice-Hall (ISBN 0-13-110163-3), available in soft-cover. The book contains a tutorial on programming in C, a description of standard C functions, a description of the UNIX operating system interface to C and a C reference manual. On the minus side, the book was published in 1978 so it is somewhat out of date with respect to the latest UNIX conventions.

The purpose of this manual is to describe Intersoft C, a subset of the C described in the book by Kernighan and Ritchie.

Intersoft C is based on "Small-C" written by Ron Cain, published in "Dr. Dobbs Journal of Computer Calisthenics & Orthodontia" in 1980. Intersoft has made a great number of modifications and enhancements to Mr. Cain's compiler in order to support a larger subset of the C language and increase performance.

Another manual (the Implementation manual) will accompany the compiler itself. The Implementation manual outlines the implementation specifics of your version of the Intersoft C compiler.

An editor is required to prepare C programs for compilation; one is not included with the compiler. An an aid to owners of uppercase only or otherwise restricted keyboards the Intersoft C compiler is case insensitive when recognizing keywords. The compiler also recognizes escape sequences for characters which your keyboard may not contain:

(%	is equivalent to	{	(Curly brace)
%)	is equivalent to	}	(Curly brace)
(.	is equivalent to	[	(Square bracket)
.)	is equivalent to	]	(Square bracket)
*:	is equivalent to	|	(Or bar)
*'	is equivalent to	~	(Tilde)
*)	is equivalent to	^	(Caret)

In addition to an editor you will also need an assembler to assemble the output of the C compiler. Possible assemblers are listed in the ordering information. Intersoft does not guarantee that any assemblers other than those listed will be able to assemble the output of the compiler.

REFERENCE GUIDE

A terse reference guide to Intersoft C v2.5 is given later in the manual. For a comprehensive introduction to C please refer to the tutorial in the book by Kernighan and Ritchie mentioned in the introduction.

The Basic Components of the C Language

Every C program is composed of global data and a number of functions which may call each other or themselves. These functions may contain local data which is allocated from the machine's stack upon each entry to the function and exists only until the function returns to the calling function. Every program must include a function with the name "main"; this is where execution begins. When the "main" function is exited the program terminates, closing all open files.

The Preprocessor

The compiler preprocesses all programs before compiling them. The primary functions of the preprocessor are:

1) Skip all comments in the program. All text enclosed by "/*" and "*/" (except where the "/*" or "*/" occurs within a string constant) is considered to be a comment.
2) Allow the use of unparameterized macros.
3) Allow the inclusion of other source files.
4) Allow conditional compilation.
5) Allow the inclusion of assembler code.

The preprocessor will be described in detail later on in this section of the manual. For now it is only important to know that all text enclosed by /* and */ constitutes a comment.

Declaring Data

Intersoft C supports two basic data types, "char" and "int". "char" variables are eight bit unsigned integers, typically used for storing characters. "int" variables are sixteen bit signed integers, typically used for storing counters and the like.

In addition to the two basic types, pointers to the basic types and single dimensional arrays of the basic types are supported. Both of these data types must be declared in terms of one of the basic types (Eg. pointer to "char" or array of "int"). The more advanced data types defined in the C language such as "structs", "unions", "typedefs", multi-dimensional arrays, "floats" and "longs" are not yet supported by Intersoft C.

Intersoft C does not support "casts". Data type conversion is performed automatically by the compiler as required. When converting an eight bit value to a sixteen bit value the original eight bit value becomes the lower eight bits of the sixteen bit value, the upper eight bits of the sixteen bit value are set to zero. When converting a sixteen bit value to an eight bit value only the lower eight bits of the sixteen bit value are used, the upper eight bits of the sixteen bit value are discarded.

Variable names are composed of alphanumeric characters and the underscore. The first character in a variable name must be either alphabetic or the underscore. The Intersoft C compiler is case sensitive for local variables. The first eight characters in local variable names are significant. The number of significant characters in global variables (or function names) is dictated by the assembler to be used; the Implementation manual will give this value. Since most assemblers are not case sensitive all global variables are turned into uppercase by the compiler. Your assembler may place further restrictions on global variable names. For example some assemblers do not allow register names to be used as variable names.

Some examples of data declarations:

```
char c;              /* Define a character variable named "c" */

char *p;             /* Define a pointer-to-character "p" */

char carray[8+2];    /* Define an array of 10 characters  */
                     /* named "carray" */

char c, *p,
     carray[8+2]; /* Define all three data items at once. */


int i;               /* Define an integer named "i" */

int *p;              /* Define "p" as a pointer-to-integer */

int iarray[8+2];     /* Define an array of integers "iarray" */
```

```
int i, *p,
    iarray[8+2];  /* Define all three data items at once. */
```

Expressions

An "expression" is a phrase in the C language which yields a numeric value. Simple constants or variables are expressions, as are constants and variables combined via "operators" (Eg. a + 3).

A special sub-class of expressions refer to manipulable pieces of storage. Elements of this sub-class are called "lvalue"(s) by Kernighan and Ritchie. Their specialty arises from the fact that only lvalues may be the target of an assignment via the assignment operators or the "++" or "--" operators. The simplest form of an lvalue is a variable. Another example is *3 which refers to the 16 bits of storage at address 3. Addresses are not lvalues, pointers may be lvalues.

Constants

There are three different types of constants, numeric, character and string.

Numeric constants may be either decimal, octal or hexidecimal. The default base for numeric constants is decimal. Octal constants are always written with a preceeding zero (Eg. 010 is octal 10 or decimal 8; NOT decimal 10!). Hexidecimal constants are always written with a leading "0x" or "0X" (Eg. 0x10 [or 0X10] is hexidecimal 10 or decimal 16; NOT decimal 10!).

C provides the ability to include certain non-graphic characters in character or string constants by "escaping" them. A character is "escaped" by immediately preceeding it with the escape character. The escape character is normally the backslash (\) however it is possible (via the "esc=" option) to have any character be considered the escape character (see the section titled "How to Use the Compiler"). The following table outlines the escape sequences supported by Intersoft C.

newline	NL (RS)	\n or \N
horizontal tab	HT	\t or \T
backspace	BS	\b or \B
carriage return	CR	\r or \R
form feed	FF	\f or \F
backslash	\	\\
single quote	'	\'
any bit pattern	ddd	\ddd

The escape sequence \ddd contains one, two or three octal digits which are taken to be the numeric value for the escaped character. Within strings the double quote character " is

escaped via the escape sequence \". It is also possible to split a long string constant across more than one line by ending the line with a backslash (\), inside a string constant both the backslash and the following newline character will be ignored. As a final note be warned that if the compiler finds a backslash which is not part of any escape sequence it can decipher the backslash will be removed from the constant or string!

PLEASE NOTE that the newline character ('\n') is actually RS (record separator, value 30) not LF (linefeed, value 10). We have found it necessary to redefine the newline character due to differences in how end of line is handled by different operating systems.

Character constants consist of a single (or escaped) character surrounded by apostrophes (single quotes '). The value of a character constant is the numeric value of the character between the apostrophes.

String constants consist of a sequence of characters (or escaped characters) surrounded by quotes (double quotes "). The value of a string constant is the address of the first memory location that contains the string constant. When the compiler stores string constants it appends an ascii NUL character ('\0').

Operators

Operators are special symbols which cause data to be manipulated. C has five classes of operators; primary operators, unary operators, binary operators, assignment operators and the comma operator.

Each operator has a "priority", higher priority operators are executed first. An example of this is the expression "a + b * c". Because multiplication has a higher priority than addition "b" will be multiplied by "c" and then the result will be added to "a". If addition had a higher "priority" then one would expect that "a" would be added to "b", then the result multiplied by "c". It is possible to alter the normal order of evaluation through the use of parentheses "()". If an expression is surrounded by parentheses it is evaluated as a unit before its value is used by operators outside the parentheses (Eg. a * (b + c) will add "b" and "c", then multiply "a" by the result).

The operands of operators are not necessarily evaluated from left to right. Nor are sub-expressions necessarily evaluated from left to right. This can cause problems if one uses the "++" or "--" operators to cause side effects within an expression. As a rule it is safe to use a variable once only in an expression if the "++" or "--" operators are used with that variable. For example the result of the expression "*ptr++ + *ptr + 1" is indeterminate since it is not guaranteed that "*ptr++" is evaluated first. Another indeterminate example is "*ptr++ = *ptr + 1". The expression "*ptr1++ = *ptr2++" is determinate since each variable only occurs once.

Primary operators

Primary operators have the highest priority. Intersoft C currently supports the () and [] primary operators. They are used for function calling and array indexing respectively.

Further Examples:

```
int a, b, array[4];

fred(a, b);     /* Calls function "fred" with arguments */
                /* "a" and "b". */

5(a, b);        /* Calls the function at address 5 with */
                /* arguments "a" and "b". */

array[3];       /* Gets the fourth element of "array" */
```

Unary Operators

Unary operators have higher priority than any binary operator but lower priority than any primary operator. All unary operators have the same priority. A list of the unary operators is:

* - The indirection operator

This operator is used to get a value from memory given a pointer to the memory address you want. Eg:

```
char *cptr, c;

cptr = 0x1000;
c = *cptr;
*cptr = c + 1;
```

Will get the character from address 1000 (Hex), assign it to the character "c", increment it then store it back into address 1000 (HEX). If the "*" operator is used with a non-pointer then the type returned is always "int". All data items are converted to 16-bit integers if indirection is used more than once. "**cptr" refers to an integer even if "cptr" is declared to be a pointer to a character.

& - The "address of" operator

This operator returns the address of an object. The "object" must have an address (must be an "lvalue"). &6 and &(a+b) are illegal. Eg:

```
char c, *cptr;

cptr = &c;
```

Will get the address of "c" and assign it to the character pointer "cptr".

\- - The unary minus operator.

This operator negates its operand, Eg: x = -x negates "x".

! - The logical negation operator.

The result of applying this operator is 1 if its operand was zero, otherwise the result is 0. Eg:

```
int x, y;

x = 3;
y = !x;
```

Assigns the value 0 to "y".

~ - The one's complement operator.

The result of applying this operator is the one's complement of its operand. The one's complement is obtained by reversing the sense of each bit in the operand. The bit pattern 10101010 would be changed to 01010101 by applying this operator.

++ - The pre- or post-increment operator.

The pre-increment operator increments its operand before fetching it for use in the remainder of the expression. The post-increment operator increments its operand after fetching it for use in the remainder of the expression. The operand must be an "lvalue", the result of the expression is not an "lvalue". Eg:

```
int a, b, c;

a = 2;
b = ++a;
c = a++;
```

Assigns the value 3 to "b" and "c" and leaves the value of "a" at 4.

-- - The pre- or post-decrement operator.

These operators behave exactly as the ++ operators mentioned above with the exception that the -- operators decrement their operand.

Binary Operators

Binary operators require two operands. The coding format is "operand1 operator operand2". All binary operators have lower priority than the unary operators. The following table shows the priorities of the binary operators in decreasing order, operators on the same line have the same priority.

```
*     /     %
+     -
>>    <<
<     >     <=     >=
==    !=
&
^
|
&&
||
?:
```

Descriptions of the binary operators:

* - The multiplication operator.

/ - The division operator.

% - The modulo (remainder) operator.

The result of applying this operator is the remainder after division, Eg. 13 % 10 is 3. The formal definition is that (a/b)*b-a = a%b where a and b are integers.

+ - The addition operator.

- - The subtraction operator.

>> - The right shift operator.

The result of this operator when applied to "a >> b" is "a" being shifted right by "b" bits. "b" may be negative, in which case "a" is shifted left the appropriate number of bits. 0 bits are shifted into the value.

<< - The left shift operator.

The result of this operator when applied to "a << b" is "a" being shifted left by "b" bits. "b" may be negative, in which case "a" is shifted right the appropriate number of bits. 0 bits are shifted into the value.

< - The "less than" operator.

"a < b" yields 1 if "a" is less than "b", otherwise it yields 0.

> - The "greater than" operator.

"a > b" yields 1 if "a" is greater than "b", otherwise it yields 0.

<= - The "less than or equal to" operator.

"a <= b" yields 1 if "a" is less than or equal to "b", otherwise it yields 0.

>= - The "greater than or equal to" operator.

"a >= b" yields 1 if "a" is greater than or equal to "b", otherwise it yields 0.

== - The "equal to" operator.

"a == b" yields 1 if "a" is equal to "b", otherwise it yields 0.

!= - The "not equal to" operator.

"a != b" yields 1 if "a" is not equal to "b", otherwise it yields 0.

& - The arithmetic "and" operator.

The result of applying this operator is a bitwise ANDing of the operands. Eg. ANDing the bit patterns 10101010 and 11001100 yields 10001000.

^ - The arithmetic "exclusive or" operator.

The result of applying this operator is a bitwise eXclusive ORing of the operands. Eg. eXclusive ORing the bit patterns 10101010 and 11001100 yields 01100110.

| - The arithmetic "or" operator.

The result of applying this operator is a bitwise ORing of the operands. Eg. ORing the bit patterns 10101010 and 11001100 yields 11101110.

&& - The logical "and" operator.

The result of applying this operator is the value of the right operand if the value of the left operand is non-zero, otherwise the result is zero and the right operand is not evaluated. This definition is slightly different from that given in the book by Kernighan and Ritchie, they define the result of applying this operator as 1 if both the operands are non-zero. The Intersoft definition does not guarantee a result of 1 but it does guarantee a non-zero result if both the operands are non-zero.

|| - The logical "or" operator.

The result of applying this operator is the value of the left operand if it is non-zero (in which case the right operand is not evaluated), otherwise the result is the value of the right operand. This definition is slightly different from that given in the book by Kernighan and Ritchie, they define the result of applying this operator as 1 if either of the operands are non-zero. The Intersoft definition does not guarantee a result of 1 but it does guarantee a non-zero result if either of the operands are non-zero.

?: - The conditional operator.

This operator is special in that it takes three operands even though it is classified as a binary operator. The form is "operand1 ? operand2 : operand3". Operand1 is always evaluated. If it is non-zero then operand2 is evaluated and is the result of the expression, otherwise operand3 is evaluated and is the result of the expression.

Assignment Operators

The assignment operators are a special class of binary operators. These operators cause the value of their left operand to be changed. All these operators have lower priority than any other binary operator (except the comma operator) and all assignment operators have the same priority.

= - Simple assignment.

This operator causes the value of its left operand to be changed to the value of its right operand. The left operand must be assignable, Eg: 3=4 is illegal but *3=4 is legal. The first example attempts to assign to the constant "3", the second example assigns to the contents of memory at address 3.

+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=
- Special assignment

The expression "a <op>= b" is functionally equivalent to
"a = (a) <op> (b)" where <op> is any of the binary operators +, -, *, /, %, >>, <<, &, ^ or |. However these constructs generate more efficient code since

the operand "a" is evaluated only once. This may seem a picky point when assigning a value to a simple variable, but consider the saving when assigning to an object that may be reached only via indirection. Eg:

```
*(*(*(ptr+a)+b)+c) *= 2;
```

is equivalent to:

```
*(*(*(ptr+a)+b)+c) = *(*(*(ptr+a)+b)+c) * 2;
```

The savings in bytes of code generated, execution time and ease of coding is significant.

The Comma Operator

The comma operator has the lowest priority of all operators. The form of an expression using the comma operator is "a, b" where "a" and "b" are expressions. The result of the operator is the value of expression "b". The comma operator is not valid within a function call argument list unless it is surrounded by parentheses. Eg: fred(a, (i=3, i+4), c). In this example the value of the second argument to "fred" is 7. The most useful place for the comma operator is in the "for" statement to allow more than one variable to be initialized or changed in the initialization or incrementation portions of the statement.

Constant Expressions

Constant expressions are a special subclass of expressions. A constant expression is an expression containing only numerical or character constants, possibly combined with operators. Valid operators for constant expressions are:

Binary: *, /, %, +, -, >>, <<, >, <, >=, <=, ==, !=, &, |, ^

Unary: -, ~, !

Ternary: ?:

Exploiting Constant Folding:

The compiler will evaluate constant sub-expressions at compile time whenever it can (producing smaller and faster code). There are two methods to improve the compiler's ability to evaluate constant sub-expressions at compile time:

1) Place constant sub-expressions within brackets ("(" and ")").
2) Place constant sub-expressions at the beginning of expressions involving both constants and variables.

Examples of constant folding:

```
3+7+i;

i+3+7;

i+(3+7);
```

In the first and third examples the compiler will calculate the value of 3+7 at compile time, and generate code to calculate i+10 instead of i+3+7.

Statements

The simplest form of a statement in C is an expression followed by a semi-colon. This form of statement is used quite frequently to assign values to variables and to perform function calls. A semicolon by itself is a valid null statement. The null statement is typically used to satisfy the syntactic rules of C when the program does not need a statement in a location which must have a statement.

Another form of a statement is a series of statements surrounded by curly braces "{}". This compound statement is similar to the BEGIN...END structure of Pascal. Compound statements are used to structure blocks of statements within functions, loops and conditional structures.

if (<expression>) <statement>

The "if" structure is a conditional which causes "<statement>" to be executed if "<expression>" is non-zero. Eg:

```
int flag, a, b;

if (flag)
  a = b;
```

Will execute "a = b" if "flag" is non-zero.

if (<expression>) <statement1> else <statement2>

The "if...else" structure is a conditional which causes "<statement1>" to be executed if "<expression>" is non-zero, otherwise "<statement2>" is executed. An "else" clause always refers to the most recent "if" statement. Note that if "<statement1>" or "<statement2>" are simple statements then they must be terminated by semi-colons. Eg:

```
int a, b, c;

if (b < c)
  a = b;
else
  a = c;
```

Will execute "a = b" if "b" is less than "c", otherwise "a = c" will be executed.

for (<expression1>;<expression2>;<expression3>) <statement>

The "for" structure is a very general purpose looping statement. "<expression1>" is evaluated once, upon entry to the "for" statement. A common use is to initialize counters. "<expression2>" is the test condition, it is evaluated each time before "<statement>" is executed. As long as "<expression2>" is non-zero "<statement>" will be executed. "<expression3>" is evaluated each time after "<statement>" is executed but before <expression2> is evaluated. A common use is to increment counters. Each of "<expression1>", "<expression2>" and "<expression3>" are optional. Omitting "<expression2>" will create an infinite loop. Eg:

```
int i, array[MAXINDEX];

for (i = 0; i < MAXINDEX; array[i++] = 0);

for (i = 0; i < MAXINDEX; i++)
  array[i] = 0;

for (;;) {
  waitforstuff();
  dostuff();
}
```

The first and second examples initialize "array" to contain zeroes. The third example loops forever calling functions "waitforstuff" and "dostuff".

while (<expression>) <statement>

The "while" construct executes "<statement>" as long as "<expression>" is non-zero. The test to see if "<expression>" is non-zero occurs before "<statement>" is executed. Eg:

```
char *ptr;

ptr = "Hi there cutie";
while (*ptr)
  processchar(*ptr++);
```

The example calls the function "processchar" with each character of the string "Hi there cutie" in succession but not the '\0' string terminator.

do <statement> while (<expression>);

The "do...while" structure is a looping construct similar to the "while" structure except that "<expression>" is tested after "<statement>" is executed. Eg:

```
char *cptr;

do {
  *cptr = getchar();
} while (*cptr++ != '\n');
*cptr = NULL;
```

The example reads a line of characters, including the end of line character ('\n'), from the standard input file into the string "cptr".

switch (<expr>) <statement>

case <constant.expr> :

default :

The "switch" structure is a multi-way "if". "<statement>" is a compound statement in which some of the sub-statements contain labels of the form:

case <constant.expr> :

These labels mark the points to which control will be transferred if "<expr>" matches the indicated "<constant.expr>". "<constant.expr>" is an expression involving only constants and operators. If a given constant label occurs more than once only the first occurence will ever be executed. Intersoft C allows up to 128 case labels within a "switch" structure. Please note that once control has been transferred to a case label execution will continue from that point through the remainder of the switch structure unless diverted via one of the "break", "continue" or "return" statements. There is a special label "default :". Control is transferred to this label if the value of "<expr>" matches none of the case constants. The case constants are compared in order of their occurence, so for efficiency's sake cases should be arranged in decreasing order of frequency of use. The "switch" structure implemented in this manner is not as fast as a jump table but it allows a much wider range of case constants. Eg:

```
int i, j, k;

switch (i) {
  case 0 :        /* If "i" is 0 or -1 then "i" is assigned to "j" */
  case -1 :
    j = i;
    break;        /* Exit the switch structure! */

  case 32767 :    /* If i is 32767 then */
    k = i;        /* Assign "i" to "k" and fall through to the */
                  /* next case! */
  case 10000 :    /* If i was originally 10000 (or 32767) then */
    putdec(k);    /* Print "k" on the standard output file */
    break;        /* Exit the switch structure! */

  default :       /* If "i" was neither 0, -1, 32767 nor 10000 */
    k = 0;        /* then assign 0 to "k" and leave the switch */
}
```

break;

The "break" statement terminates the execution of the smallest enclosing "for", "while", "do...while" or "switch". The next statement executed is the first statement following the "terminated" statement. The "break" statement is valid only within these statements. Eg:

```
for (;;) {
  wait();
  dostuff();
  if (done)
    break;
}
```

In the above example when the variable "done" is non-zero the infinite "for" loop is terminated.

continue;

The "continue" statement causes program execution to be transferred to the next iteration of a "for", "while" or "do...while" loop. "continue" is equivalent to a "goto" whose target is after the last executable statement in the loop but is still inside the loop. Note: Intersoft C does not support the "goto" construct, its mention was for illustration only. Eg:

```
for (;;) {
  if (!wait())
    continue;     /* If wait() failed then re-loop */
  dostuff();
}
```

```
for (;;) {
  if (!wait())
    goto contin;  /* This is equivalent to the previous */
  dostuff();      /* example.  Intersoft C DOES NOT */
contin :          /* SUPPORT the "goto" */
}
```

return <expression> ;

The return statement causes program execution to be transferred back to the calling function. If "<expression>" is present then a single integer value may be returned, "<expression>" is not mandatory. It is not necessary to put a "return" statement at the end of a function unless a value is to be returned, functions return automatically after executing their last statement.

Functions

To declare a function one writes its name, a list of its arguments surrounded by brackets then the types of its arguments. A single simple or compound statement composes the function. If local variables are required they must be defined before any statements are written. Eg:

```
max(a, b)           /* The argument list declares the calling */
  int a, b;         /* sequence. Arguments must also be */
                    /* declared before the body of the function! */
{                   /* A compound statement */
  if (a > b)
    return a;
  else
    return b;
}
```

```
max(a, b)
  int a, b;
  return (a > b) ? a : b;  /* A simple statement */
```

```
longpause()         /* No arguments */
{                   /* A compound statement */
  int i, j;         /* Local variables */

  for (i = 0; i < 32767; ++i)
    for (j = 0; j < 32767; ++j);
}
```

A function call is an expression. It is possible to call a function by its address rather than by its name, or to call computed addresses. Eg:

```
int fptr, a, b, i;

func(a, b);         /* Call "func", discard the return value */
fptr = func;        /* Get the address of the function */
fptr(a, b);         /* Call "func", discard the return value */
i = fptr(a, b);     /* Uses the return value */

(0xff)(a);          /* Call a function at address 0xff */
```

The Preprocessor Re-visited

All C programs are preprocessed before they are compiled. The preprocessor changes long strings of whitespace characters into single spaces and skips comments. A comment is defined as all text between enclosing "/*" and "*/" symbols. Comments are not nestable.

In addition to these basic functions there are several preprocessor commands which are very useful to programmers. All preprocessor commands must be written with the hash symbol (#) in column 1 of the source file.

Macros

It is possible to declare unparameterized macros through the use of the #define preprocessor command. The format of this command is:

#define <identifier> <token string>

"<identifier>" is any valid identifier (variable name), it is the name of the macro. "<token string>" is any string of printable ascii text including whitespace up to the end of the source line. From the statement following the macro definition onward whenever "<identifier>" occurs in the source file it will be replaced by "<token string>". It is possible to use an unparameterized macro in the definition of another unparamaterized macro. The ability to define unparameterized macros is useful since it lets programmers define symbolic names for constants at the top of a module and use the constants throughout the module.

Inclusion of other Source files

It is possible to instruct the preprocessor to switch input from the current source file to some other specified file. When the other file has been compiled the preprocessor automatically switches back to the old source file. There are two forms of this directive:

```
#include "filename"
or
#include <filename>
```

In both cases filename must be a valid file name to your operating system.

Conditional Compilation

The preprocessor has several nestable constructs to facilitate conditional compilation.

#ifdef <identifier>

Will check to see if "<identifier>" has been defined via "#define". If so then the code following the "#ifdef" will be compiled until the corresponding "#else" or "#endif" is encountered. If "<identifier>" wasn't defined then no code will be compiled until the corresponding "#else" or "#endif" is encountered.

#ifndef <identifier>

Is similar to the "#ifdef" construct but will generate the code following it if "<identifier>" is not defined.

#else

This is the "else" clause for either the "#ifdef" or "#ifndef" constructs. It is optional.

#endif

This terminates a "#ifdef" or "#ifndef" construct. It is mandatory.

Example 1:

```
#define FLAG1 0

#ifdef FLAG1
  printf("FLAG1 is defined\n");
#endif
```

Example 2:

```
#ifndef FLAG1
  printf("FLAG1 is not defined\n");
#endif
```

Example 3:

```
/* This example prints a message to tell whether or
 * not FLAG1 is defined.
 */
#ifdef FLAG1
  printf("FLAG1 is defined\n");
#else
  printf("FLAG1 is not defined\n");
#endif
```

Example 4:

```
/* This example also prints a message to tell whether
 * or not FLAG1 is defined.
 */
#ifndef FLAG1
  printf("FLAG1 is not defined\n");
#else
  printf("FLAG1 is defined\n");
#endif
```

Example 5:

```
/* This example prints a message to tell whether
 * or not FLAG1 and FLAG2 are defined.  It illustrates
 * the ability to nest the conditional compilation directives.
 */
#ifdef FLAG1
#ifdef FLAG2
  printf("FLAG1 is defined, FLAG2 is defined\n");
#else
  printf("FLAG1 is defined, FLAG2 is not defined\n");
#endif
#else
#ifdef FLAG2
  printf("FLAG1 is not defined, FLAG2 is defined\n");
#else
  printf("FLAG1 is not defined, FLAG2 is not defined\n");
#endif
#endif
```

Inclusion of Assembly Code

It is possible to include assembly code in C programs via the "#asm" and "#endasm" preprocessor commands. Text between the "#asm" and "#endasm" directives is copied directly to the assembler output file. No "#define" macro substitution, source file inclusion or conditional compilation is possible within a

"#asm" construct. The details of how to write assembly code to interface with C code will be explained in the Implementation manual for your compiler. It is not explained here because the details differ for different machine architectures.

The ability to put assembly code within a higher level language program is useful in time critical situations or when the program must interface to software written in another language.

I/O IN C

This section deals with performing i/o in the C language. C contains no built-in i/o facilities. I/O is performed via run-time support functions. An entire section of this manual (titled "Run Time Support Functions") is dedicated to providing a quick reference to the support functions included with the compiler. I/O functions are only one class of the supplied functions.

Every C program has access to three default files (more files may be opened and closed as needed). These files are the standard input, output and error files. The default files are opened automatically before a C program is executed and are closed automatically when the "main" function returns or when the function "exit" is called. The default files are all character oriented ascii text files.

The standard input file may only be read. This file corresponds to buffered console input unless reassigned. To reassign the standard input file type "<filename" on the command line ("filename" is the name of the file or device to be used as the standard input file).

The standard output file may only be written. This file corresponds to output on the system console unless reassigned. To reassign the standard output file type ">filename" or ">>filename" on the command line. ">filename" specifies that "filename" will be created and written (or re-written, if the file already existed). ">>filename" specifies that the standard output file of the program will be appended to "filename".

The standard error file corresponds to the system console and may not be reassigned via the command line. This file is normally used for messages which should always be written to the system console.

All files are accessed via their file pointer, which is a variable of the type pointer to "FILE" or a constant. "FILE" is an unparamaterized macro defined to be "int" for this release, it may change in future releases so it is prudent to define file pointers to be of type pointer to "FILE". The file pointers (sometimes referred to as "unit" numbers) for the default files are constants with the following names:

```
input file  -- "stdin"  or "STDIN"
output file -- "stdout" or "STDOUT"
error file  -- "stderr" or "STDERR"
```

The constants "FILE", "stdin", etc. are defined in a file provided with the compiler. Please refer to the Implementation manual for the name of the constant file for your machine.

None of the i/o routines are guaranteed to be re-entrant, they access global data and make calls to the operating system. Programmers wishing to perform i/o from interrupt handlers are

advised to be cautious.

getchar() and putchar()

getchar() and putchar() are two simple functions to read from the standard input file and write to the standard output file. The function getchar() takes no arguments, it returns the next character from the standard input file or the constant EOF upon end of file (or an i/o error). The constant EOF is a 16-bit integer, not an 8-bit character. If a program puts the value returned by getchar() (or putchar()) into an 8-bit variable and later compares it to the constant EOF the two will NEVER be equal. The function putchar(c) takes one argument (the character to be written) and returns the character written or EOF upon an i/o error.

A simple example of a program that copies the standard input file to the standard output file (a more complex example is the copy program given in the section titled "Sample C Programs") is:

```
main() {
  int i;

  while ((i = getchar()) != EOF)
    putchar(i);
}
```

gets() and puts()

It is frequently more convenient to perform i/o in blocks larger than a single character. "gets(s)" reads a line from the standard input file (to the first newline character) and stores it in the string at address "s" (appending a '\0' (ascii NUL) as a string terminator). The function returns the address "s" or EOF upon end of file or an i/o error. "puts(s)" writes the string at address "s" to the standard output file. The string terminator ('\0') is not written. The function returns the address "s" upon success or EOF upon failure. A simple program that copies the standard input file to the standard output file is:

```
main() {
  char line[512];

  while (gets(line) != EOF) {
    puts(line);
    putchar('\n');
  }
}
```

getdec() and outdec()

These functions convert integers (8-bit "char" or 16-bit "int" types) from their binary form to ascii. "getdec()" takes no arguments and returns an integer, the result of reading and converting an ascii string like "-300" or " 2". "outdec(n)" converts the integer "n" to ascii and writes it to the standard input file. Example:

```
main() {
  char line[512];
  int i;

  puts("Please enter an integer ? ");
  i = getdec();
  puts("The integer was ");
  outdec(i);
  puts("\n");
}
```

error()

"error(s,t)" writes the strings at the addresses "s" and "t" to the standard error file, then exits the program closing all open files.

Example 1:

```
char *fname;

error("Can't open file ", fname);
```

Example 2:

```
main(argc, argv)
  int argc, *argv;
{
  int n;

  /* .
     .
     .
   */
  error("Illegal option ", argv[n]);
}
```

Example 3:

```
/* This example shows a null string as the second argument */
error("Insufficient memory", "");
```

Opening and Closing Additional Text Files

The function "fopen(fname,mode)" attempts to open a file and returns the file pointer for the file or zero if the file could not be opened. "fname" is the address of a string containing the name of the file to be opened. "mode" is the address of a string indicating how the file is to be accessed:

```
"r" or "R"  --  Read only
"w" or "W"  --  Write only
"a" or "A"  --  Append only
```

Example:

```
/* This opens "fname" for append or kills the program */
openapp(fname)
  char *fname;
{
  FILE *fp;

  fp = fopen(fname, "a");
  if (!fp)
    error("Can't open", fname);
  return fp;
}
```

The function "ckopen(fname,mode)" takes the same arguments as the function "fopen" but always returns the file pointer of an open file. If the file could not be opened the error message

"Can't open <file>" is written (<file> is the string at address "fname") and the program is exited, closing all open files. The following example will either open the file "file" for read access or exit the program:

```
FILE *fp;

fp = ckopen("file", "r")
```

The function "fclose(fp)" closes the file associated with the file pointer "fp". The function returns non-zero upon success or zero upon failure. The following example opens then closes a file called "file":

```
fclose(fopen("file", "w"));
```

I/O with non-Default Files

The functions getchar(), putchar(), gets(), puts(), getdec() and putdec() have equivalents which perform the same functions and return the same values but take an additional argument, the file pointer of the file to be accessed:

```
getchar()   --  getc(fp)
putchar(c)  --  putc(c,fp)
gets(s)     --  fgets(s,fp)
puts(s)     --  fputs(s,fp)
getdec()    --  getnum(fp)
putdec(n)   --  putnum(n,fp)
```

ungetc()

The function "ungetc(c,fp)" unreads a single character "c" from the file "fp". Only one character may be unread at a time.

feof()

The function "feof(fp)" returns non-zero if end of file has been reached on the file "fp" or if "fp" does not represent an open file.

Formatted I/O

Several functions are available to perform formatted i/o. There are two classes; input oriented functions and output oriented functions. Detailed descriptions of the calling sequences of these functions (and examples of their use) are given in the section titled "Run Time Support Functions".

"scanf(control, arg, arg, ...)" uses the "control" argument as a format specifier, the function reads from the standard input file. The function "fscanf(fp, control, arg, arg, ...)" is the same as "scanf" but takes an extra argument, the file pointer of the file to be read. The function "sscanf(ptr, control, arg, arg, ...)" is the same as "scanf" but takes an extra argument, the address of the string in memory to be read.

"printf(control, arg, arg, ...)" uses the "control" argument as a format specifier, the function writes to the standard output file. The function "fprintf(fp, control, arg, arg, ...)" is the same as "printf" but takes an extra argument, the file pointer of the file to be written. The function "sprintf(ptr, control, arg, arg, ...)" is the same as "printf" but takes an extra argument, the address of the string to be written.

I/O with Binary Files

The run time support library currently contains no functions to support i/o with binary files. If these functions are required they may be written and added to the run time support library or kept in a library of your own. The interface to binary files is defined in chapter eight of the book by Kernighan and Ritchie. The most useful functions would be open(), close(), read(), write() and seek().

THE COMMAND LINE

It is possible to pass information to your C programs via the command line which invokes the program. The Intersoft run time support provides C programs with a tokenized copy of the command line if it is desired. To get at the copy of the command line from C programs all that you need to do is declare the "main" function of a program to have two arguments, as follows:

```
main(argc, argv)
  int argc, argv;
{
  /* The body of the main procedure */
}
```

"argc" is a count of the number of tokens (a token is a string of non-whitespace characters) on the command line, including the name of the program itself. "argv" is a pointer to an array of pointers to strings. These strings are the individual tokens from the command line, there are "argc" of them. We declare it to be an integer because Intersoft C does not support the data type pointer-to-pointer.

According to UNIX conventions argv[0] is a pointer to a string containing the name of the program being executed. Unfortunately some micro-computer operating systems make the name of the program very hard to access. You may find that argv[0] contains a pointer to a null string.

Some examples of programs that use the command line to pass information from the operator are given in the section titled "Sample C Programs".

I/O Re-direction

Another very useful feature of Intersoft C is the ability to re-direct the standard input and output files of a program to any file or device from the command line.

By placing ">file" anywhere on the command line after the program name the operator specifies that the standard output file is to go to "file". By placing ">>file" on the command line the operator specifies that the standard output file is to be appended to "file". "file" may be either a disk file or one of the devices supported by Intersoft C. Please refer to the Implementation manual for a list of the devices for your micro, devices vary from computer to computer.

Similarly by placing "<file" anywhere on the command line after the program name the operator specifies that the standard input file is to be taken from "file".

Please refer to the copy command in the section titled "Sample C Programs" for an example of the uses of i/o re-direction.

SAMPLE C PROGRAMS

Please note that all listings in this section are of the CP/M versions of these programs. The format of the file names in the #include directives may be different for your system.

The Standard Header File

The following is a listing of the standard i/o header file. It illustrates how constants can be grouped together via the "#define" preprocessor command.

```
/*
 * The Intersoft C V2.5 standard i/o definitions.
 */
#define NULL 0
#define FORMFEED 12
#define TRUE -1
#define FALSE 0

#define FILE int
#define EOL  30
#define EOS   0
#define EOF  -1
#define STDIN  1
#define STDOUT 2
#define STDERR 3
#define stdin  1
#define stdout 2
#define stderr 3
```

A Program to Split Source Libraries

The following listing is of a program used at Intersoft to help maintain source libraries. Source libraries look very much like normal C programs. The difference is that the beginning of each module is marked with a line beginning with an exclamation mark. The remainder of that line is taken to be the module name by this program. A source library will not compile correctly until it is split because of these module marking source lines. The source library may be in any language.

This program illustrates how command line arguments may be processed; see the function "main".

Take special note of the function "amatch", it gives an example of a very efficient way of sequentially processing strings. It is possible to use array indexing to achieve the same effect, but using the ++ and -- operators on pointers to the strings generates more efficient code.

```
/*
 * This program splits a source library up into individual
 * modules.  It also produces a list of the names of the modules
 * in the library that was just split.
 *
 * Copyright (c) 1982 by Intersoft
 *
 * by: Richard B. McMurray
 */

#include <stdio.h>

#define LINELEN 130
#define MERGECH '%'
#define MAXMODS 100

char line[LINELEN];     /* A source line */
FILE *infile,           /* The input file pointer */
     *outfile;          /* The output file pointer */
int  modules[MAXMODS],  /* Addresses of the strings containing */
                        /* the module names */
     nmods;             /* The number of modules */

main(argc, argv)
  int  argc, *argv;
{
  int  i;
  char c;

  /* Verify the number of arguments, initialize data */
```

```
  if (argc > 1)
    usage();
  outfile = nmods = 0;

  /* Skip lines in the source library until a module marker
   * or end of file is encountered
   */
  while (notnewfile());

  /* Loop until a module marker is not encountered */
  for (;*line == '!';) {

    /* Open a new output module, closing the old one */
    openout();

    /* Copy a portion of the source library (to the next module
     * marker) into the output module
     */
    while (notnewfile()) {
      fputs(line, outfile);
      putc('\n', outfile);
    }
  }

  /* Write the list of module names, one per line */
  for (i = 0; i < nmods; ++i) {
    puts(modules[i]);
    putchar('\n');
  }
}


/* This function prints the USAGE message and exits. */
usage(){
  puts("USAGE: split <infile >namelist\n");
  exit(1);
}


/* This function performs an anchored string match. */
/* "s" is assumed to be as long as or longer than "t". */
amatch(s, t)
  char *s, *t;
{
  while (*t && *s++ == *t++);
  return !*t;
}


/*
 * This function reads a line from the input library,
 * returning zero if the line specifies a new file,
```

```
 * otherwise non-zero is returned.
 */
notnewfile()
  return (gets(line) != EOF || *line) && *line != '!';



/*
 * This function closes the current output file and opens
 * the new output file.  The name of the new output file
 * is stored in the array "modules" for use later when
 * the control file to process the modules is created.
 */
openout() {
  if (outfile)
    fclose(outfile);
  modules[nmods++] = strsave(&line[1]);
  outfile = ckopen(&line[1], "w");
}



/*
 * This function saves a string dynamically and returns
 * the address of the saved copy.
 */
strsave(s)
  char *s;
{
  char *p;

  if (p = alloc(strlen(s) + 1))
    strcpy(p, s);
  return p;
}
```

A Merge Program

The following program takes a template file containing the character '%' where it expects a file name to be filled in, and the list of file names produced by the previous program. This program copies the template file once for each file name, substituting the file name for the '%' every time it occurs in the template file. These copies of the template file are concatenated together into the standard output file.

This program is used to create a command file to process each of the modules extracted from a source library by the "split" program.

```
/*
 * This program creates a control file to process each of
 * the output modules of the split program according to
 * a specified template.
 *
 * Copyright (c) 1982 by Intersoft
 *
 * by: Richard B. McMurray
 */

#include <stdio.h>

#define LINELEN 130
#define MERGECH '%'
#define TABCH 9

char line[LINELEN],  /* A source line */
     repchar;        /* The replacement character */
int  mergefile;      /* The template file unit number */

main(argc, argv)
  int  argc, *argv;
{
  char *startbuf, *endbuf, *ptr, *bsize;
  int c;

  /* Parse the command line */
  if (argc < 2 || argc > 3 || !(mergefile = fopen(argv[1], "r")))
    usage();
  repchar = MERGECH;
  if (argc == 3)
    if (!amatch(argv[2], "c="))
      usage();
    else
      repchar = fetchch(argv[2], 2);
```

```
  /* Read the template file into a memory buffer
   * Trailing whitespace is stripped from all lines of
   * the template file.  The template file is terminated
   * by either EOF or by the first non-printable,
   * non-whitespace character.
   */
  bsize = memleft();
  if (!(startbuf = alloc(bsize))) {
    puts("Insufficient memory!\n");
    exit(1);
  }
  endbuf = startbuf;
  ptr = startbuf - 1;
  while ((c = getc(mergefile)) != EOF) {
    if ((c >= ' ' && c <= '~') || c == TABCH) {
      if (endbuf >= startbuf + bsize) {
        puts("Insufficient memory to save the template file!\n");
        exit(1);
      }
      if (c != ' ' && c != TABCH)
        ptr = endbuf;
      *endbuf++ = c;
    } else
      if (c == '\n') {
        *++ptr = '\n';
        endbuf = ptr + 1;
      } else
        break;
  }
  endbuf = ptr + 1;
  fclose(mergefile);

  /* Read module names from the standard input file */
  while (gets(line) != EOF) {

    /* Module names must begin with a printable non-space
     * character.
     */
    if (*line <= ' ' || *line > '~')
      break;

    /* Copy the template file to the standard output file,
     * replacing the "repchar" with the module name
     */
    for (ptr = startbuf; ptr < endbuf; ptr++) {
      if (*ptr == repchar)
        puts(line);
      else
        putchar(*ptr);
    }
  }
}
```

```
/* This function prints the USAGE message and exits. */
usage(){
  puts("USAGE: merge template [c=x] <infile >outfile\n");
  puts("Makes n copies of template where n = # of lines in infile\n");
  puts("Uses successive lines of infile to replace all instances\n");
  puts("of the merge character from the template file.\n");
  exit(1);
}


/* This function returns the n+1th character from a string. */
fetchch(s, n)
  char *s;
  int  n;
  return s[n];


/* This function performs an anchored string match. */
/* "s" is assumed to be as long as or longer than "t". */
amatch(s, t)
  char *s, *t;
{
  while (*t && *s++ == *t++);
  return !*t;
}
```

Printf, Fprintf and Sprintf

The following listing is the header file for the printf(), fprintf(), sprintf() and ccputf() functions from the Intersoft run time library.

```
/*
 * Definitions for printf, fprintf() and sprintf()
 *
 * Copyright (c) 1982 by Intersoft
 *
 * By: R. McMurray
 *
 * Based on printf() by Mike Gore.
 */

#define STRING   0
#define NUMBER   1
#define ZERO     '0'
#define BLANK    ' '
#define LEFT     0
#define RIGHT    1
#define PLUS     0
#define MINUS    1
#define MAXINT   32767
#define BUFSIZE  16
#define WIDTH    132

int ccnarg;      /* Hold the number of arguments to printf */
int ccarg;       /* Points to the first argument */
char *ccbptr;    /* Typically, beginning of string pointer  */
char *cceptr;    /* Typically, end of string pointer */
int  ccunout;    /* The output unit number, 0 for sprintf() */
char *ccsadr;    /* The string address for sprintf() */
```

The following listing is a source library containing four of the Intersoft run time support functions. Source lines beginning with "!" mark the beginning of a module. Note how the "#asm" preprocessor command is used to manipulate a variable number of arguments to these functions.

```
/*
 * printf(), fprintf(), sprintf() and ccputf() - formatted output
 *
 * Copyright (c) 1982 by Intersoft
 *
 * By: R. McMurray
 *
 * Based on printf() by Mike Gore.
 */

!PRINTF
#include <stdio.h>
#include "printf.h"

printf() {
/*
 *   All of the arguments that are passed to printf
 * are pushed on the stack. The A register contains
 * the argument count. CCARG is the address of the
 * first argument on the stack.  CCARG is the address
 * of the format string.
 */
#asm
        LD      L,A
        LD      H,0
        LD      (CCNARG),HL     ; Save # of args. in CCNARG.
        ADD     HL,HL           ; Index by words not bytes.
        ADD     HL,SP
        LD      (CCARG),HL      ; Save address of pointer to
                                ; first argument.
#endasm
  ccunout = STDOUT;     /* Set the output unit */
  ccsadr = 0;
  ccputf();             /* Common formatted output routine */
} /* end printf */


!FPRINTF
#include <stdio.h>
#include "printf.h"


fprintf() {
/*
```

```
 *   All of the arguments that are passed to fprintf
 * are pushed on the stack. The A register contains
 * the argument count. CCARG is the address of the
 * first argument on the stack.
 * CCARG is the address of the file pointer.
 */
#asm
        LD      L,A
        LD      H,0
        DEC     HL                  ; Subtract file unit arg.
        LD      (CCNARG),HL         ; Save # of args in CCNARG.
        ADD     HL,HL               ; Index by words not bytes.
        ADD     HL,SP
        LD      (CCARG),HL          ; Save address of pointer to
                                    ; first argument.
#endasm
  ccunout = *(ccarg + 2);           /* File pointer is the first arg. */
  ccsadr = 0;
  ccputf();                 /* Common formatted output routine */
} /* end fprintf */


!SPRINTF
#include <stdio.h>
#include "printf.h"


sprintf() {
/*
 *   All of the arguments that are passed to sprintf
 * are pushed on the stack. The A register contains
 * the argument count. CCARG is the address of the
 * first argument on the stack after the string address.
 * CCARG is the address of the format string.
 */
#asm
        LD      L,A
        LD      H,0
        DEC     HL                  ; Subtract string ptr arg.
        LD      (CCNARG),HL         ; Save # of args. in CCNARG.
        ADD     HL,HL               ; Index by words not bytes.
        ADD     HL,SP
        LD      (CCARG),HL          ; Save address of pointer to
                                    ; first argument.
#endasm
  ccsadr = *(ccarg + 2);
  ccunout = 0;
  ccputf();                 /* Common formatted output routine */
  ccpch(0);                 /* Terminate the string */
} /* end sprintf */
```

```
!CCPUTF
#include <stdio.h>
#include "printf.h"

ccputf() {
  int  sign;              /* Sign flag for %d */
  int  mask;              /* Mask for octal & hex numbers */
  int  nbits;             /* The number of bits in mask */
  int  num;               /* Unconverted # for %u %d %o %h */
  int  big;               /* Assigned the value MAXINT */
  int  carry;             /* Used in %u for carry */
  int  temp;              /* Used in %u */
  int  flag;              /* Number / string flag */
  int  flr;               /* Left or right justify flag */
  int  fmin;              /* Minimum field width */
  int  fmax;              /* Maximum field width */
  int  alen;              /* Length    */
  int  n;                 /* The length of a string */
  int  npad;              /* Number of pad chr chrs */
  char fcode;             /* Holds current fmt chr */
  char fill;              /* Fill character  */
  int  counter;           /* Current argument number */
  char *arg;              /* Pointer to current argument */
  char *fmt;              /* Pointer to the fmt control arg. */
  char buf[BUFSIZE];      /* Buffer for %c and numbers */

  counter = 1;
  fmt = ccargs(counter);
  if (!fmt)               /* Check for missing control arg. */
    return;

  for(;;) {
    ccbptr = fmt;
    cceptr = 0xffff;

    /* Output all non-format characters */
    while ((fcode = *ccbptr) != '%') {
      if (fcode == EOS)
        return;
      ccpch(*ccbptr++);
    }
    fmt = ++ccbptr;       /* Move to next format character */

    flr = RIGHT;          /* Right justify by default */
    if (*fmt == '-') {
      flr = LEFT;         /* Left justify if specified */
      ++fmt;
    }

    fill = BLANK;         /* Fill with blanks by default */
    if (*fmt == '0') {
      fill = ZERO;        /* Fill with zeroes if specified */
      ++fmt;
    }
```

```
    fmin = 0;   /* Get min. field width, default is 0 */
    while (isdigit(*fmt))
      fmin = fmin * 10 + *fmt++ - '0';
    if (*fmt == '.')
      ++fmt;

    fmax = 0;   /* Get max. field width, default is WIDTH */
    while (isdigit(*fmt))
      fmax = fmax * 10 + *fmt++ - '0';
    if (fmax == 0 || fmax > WIDTH)
      fmax = WIDTH;

    arg = ccargs(++counter);     /* Get arg. to print */

    /* Select the appropriate format */
    switch (fcode = toupper(*fmt++)) {

      /* Single character */
      case 'C':
        flag = NUMBER;
        ccbptr = cceptr = buf + BUFSIZE;
        *--ccbptr = arg & 0xFF;
        break;

      /* String */
      case 'S':
        flag = STRING;
        if (!arg)
          arg = "";
        ccbptr = cceptr = arg;
        n = 0;
        while (*cceptr != EOS)
          ++cceptr;
        break;

      /* Octal or Hex number */
      case 'O':
      case 'X':
        flag = NUMBER;
        num = arg;
        if (fcode == 'O') {
          mask = 0x7;
          nbits = 3;
        } else {
          mask = 0xf;
          nbits = 4;
        }
        ccbptr = cceptr = buf + BUFSIZE;
        do {

          *--ccbptr = (num & mask) + ((num & mask) < 10 ?
                      060 : 0127);
        } while ((num >>= nbits) > 0);
        break;
```

```
        /* Signed Decimal number */
        case 'D' :
          flag = NUMBER;
          num = arg;
          ccbptr = cceptr = buf + BUFSIZE;
          sign = (num < 0);
          do {
            temp = num % 10;
            if (temp < 0)
              temp = -temp;
            *--ccbptr = '0' + temp;
            num /= 10;
          } while (num);
          if (sign)
            *--ccbptr = '-';
          break;

        /* Unsigned Decimal number */
        case 'U' :
          flag = NUMBER;
          num = arg;
          ccbptr = cceptr = buf + BUFSIZE;
          if (num >= 0)
            do {
              *--ccbptr = '0' + (num % 10);
            } while ((num /= 10) > 0);
          else {
            big = MAXINT;
            num = (num & big);
            carry = 1;
            do {
              temp = (num % 10) + (big % 10) + carry;
              carry = temp / 10;
              temp %= 10;
              *--ccbptr = '0' + temp;
              num /= 10;
            } while ((big /= 10) > 0);
          }
          break;

        /* Unrecognized format */
        default :
          flag = STRING;
          cceptr = fmt;
          fmax = WIDTH;
          fmin = 0;
          break;
      } /* end switch */

      /* Enforce the maximum field width. */
      if (ccbptr + fmax < cceptr)
        cceptr = ccbptr + fmax;

      /* Propagate minus sign to the front of a number padded */
      /* with preceding zeroes. */
```

```
    if (*ccbptr == '-' && fill == ZERO && flag == NUMBER)
      ccpch(*ccbptr++);

    alen = cceptr - ccbptr;          /* Compute the min. # of */
    if (alen > fmin)                 /* characters to print */
      fmin = alen;
    npad = fmin - alen;

    if (flr == RIGHT)                /* Right justify */
      while (npad-- > 0)
        ccpch(fill);

    while (cceptr > ccbptr)          /* Output the item */
      ccpch(*ccbptr++);

    if (flr == LEFT)                 /* Left justify */
      while (npad-- > 0)
        ccpch(fill);
  } /* end for(;;) */
} /* end ccputf */


/* Returns argument number z */
ccargs(z)
  int z;
  return (!ccnarg || z > ccnarg) ? 0 : *(ccarg - z - z + 2);


/* Output a character */
ccpch(c)
  int  c;
{
  if (ccunout)
    putc(c, ccunout);
  else
    *ccsadr++ = c;
}
```

The following is a listing of a simple program that uses the i/o redirection capabilities of Intersoft C to become a useful copy program. The program copies the standard input file to the standard output file. Through the i/o redirection capabilities of Intersoft C this program can be adapted to copy files to or from virtually all devices on your computer.

Possible uses of the Copy command:

- The command "copy file1 >file2" will copy "file1" to "file2".
- The command "copy file1 >>file2" will append "file1" to "file2" if Intersoft C for your micro supports appending to files.
- The copy command can become a printer program by specifying file2 to be the printer (list device). Device names are described in the Implementation manual.
- The command "copy file1" will list file1 on the system console.
- The command "copy >file1" will allow you to enter file1 from the console.
- The command "copy >>file1" will allow you to append to file1 from the console if Intersoft C for your micro supports appending to files.

Find out what the end of file character is for your micro before attempting to use the "copy" command to enter files from the system console.

```
/*
 * Copy from standard input to standard output.
 * Copyright April 1981 by Intersoft Unlimited
 * by Bernie Roehl
 */
#include <stdio.h>
main(argc, argv)
  int argc, argv[];
{
  int in, c;
  if (argc == 1)
     in = STDIN;
  else
     in = ckopen(argv[1], "r");
  while ((c = getc(in)) != EOF)
     putc(c, STDOUT);
}
```

HOW TO USE THE COMPILER

First create a C program using a suitable editor, then execute the compiler via the command:

C <infile> [o=<outfile>] [*options]

where:

<infile> - The input file name.

o=<outfile> - The output file name.

Some versions of the compiler will automatically create an output file with the same name as the input file but with a different file name extension. Please refer to the Implementation manual for the details concerning your machine.

*options - Compiler options. May be any (or none) of:

-g - Do not define storage for global variables.

Normally the compiler will allocate storage for all global variables defined in the module being compiled.

One way of structuring large programs is as several separately compiled modules. The compiler requires that the global variables be defined in each of the modules compiled. In the C defined in the book by Kernighan and Ritchie the keyword "extern" causes global (or local) declarations to be defined for the compiler but instructs the compiler to allocate no storage for these variables. A program could then be structured so that modules which do not contain a given global variable but do access the variable could define the variable as "extern".

Intersoft C does not support the "extern" keyword. Globals must be collected into one file and included (via #include) in all modules for compilation. The "-g" option should be used on the compilation of all code modules of the program to prevent multiple copies of the global variables being allocated. Then the file containing all the globals is compiled without the "-g" option to create a module with storage for all the global variables.

-n - Do not pass the number of arguments to functions.

The compiler will normally pass the number of arguments to each function every time it is called. This option prevents the compiler from generating code to pass the number of arguments to each function called. It is always preferable not to pass the number of arguments to a function. The run time support functions printf(), fprintf(), sprintf(), scanf(), fscanf() and sscanf() require the number of arguments information. If your module uses one of these run time support functions then do not use this option. The default is to pass the number of arguments to every function called.

+s - Include the C source as comments in the output file.

This option is occasionally useful during debugging. The default is not to include the source code in the output file.

lp=nn - Set the size of the literal pool (in bytes).

The literal pool contains string constants. It is emptied after each function is compiled. In the event that a function requires a literal pool larger than the default size this option may be used. The compiler produces a diagnostic when the literal pool overflows. nn may be a decimal, hexidecimal or octal unsigned constant. The default is 256 bytes.

dp=nn - Set the size of the "#define" (macro) pool (in bytes).

The "#define" preprocessor command is explained in the section titled "Reference Guide". The compiler will produce a diagnostic if the "#define" (or macro) pool overflows. "nn" may be a decimal, hexidecimal or octal unsigned constant. The default is 1200 bytes.

#d=nn - Set the maximum number of "#define"s (macros).

The "#define" preprocessor command is explained in the section titled "Reference Guide". The compiler will produce a diagnostic if a module contains too many "#define" statements. "nn" may be a decimal, hexidecimal or octal unsigned constant. The default is 100 "#define"s.

#g=nn - Set the maximum number of global variables.

The compiler will produce a diagnostic if a module contains too many global variables. "nn" may be a decimal, hexidecimal or octal unsigned constant. The default is 160 global variables.

#l=nn - Set the maximum number of local variables.

The compiler will produce a diagnostic if a function contains too many local variables. "nn" may be a decimal, hexidecimal or octal unsigned constant. The default is 30 local variables.

ll=nn - Set the maximum length of a source line.

This value is actually one greater than the maximum length of a source line, one byte of the line is reserved for a line terminator. "nn" may be a decimal, hexidecimal or octal unsigned constant. The default is 132 bytes.

lab=nn - Set the first local label number.

The C compiler generates local labels. If you are using the C compiler in conjunction with a linking loader you never need this option. If you do not have a linking loader then you should use this option in conjunction with the "stats=nn" option to insure that local labels do not overlap in a program that consists of several separately compiled modules. The default first label number is 5000.

esc=c - Set the character escape character.

If your keyboard is not capable of producing the backslash (\) character then you may re-define the escape character via this option. Eg: if "esc=*" then '*n' is the newline character, equivalent to '\n' when "esc=\". For more on how to use escape characters refer to the section titled "Reference Guide". The default is '\'.

stats=nn - Print compiler statistics.

If nn is non-zero then the compiler will display the first and last local label number used in the compilation. This option is of interest only to those who are using the compiler in conjunction with an assembler that does not have a linking loader. Also see the "lab=nn" option. The default is zero (no statistics).

f=nn - Generate size optimized code for Z80's.

This option concerns only Z80 owners. If "nn" is one (1) then the compiler will use reset vectors 6 and 7 to access commonly used run time support functions. This actually slows programs marginally but decreases the amount of code generated for a program. The default is zero (do not use reset vectors).

NOTE: Reset vectors are 3&4 for TRSDOS & LDOS. But are 1&2 for NEWDOS 80 Ver 2.0

RUN TIME SUPPORT FUNCTIONS

The run time support library is included as a searchable library for those who have linking loaders. Please refer to the Implementation manual for the name of the library file and how to use it.

The source for the run time support libraries consists of several files. See the Implementation manual for the names of these files. Each file contains several functions. The functions are separated by lines starting with an exclamation mark and a six (or fewer) letter name. These files constitute a simple form of source library. Listings of two programs which manipulate this kind of source library are given in the section titled "Sample C Programs". The functions are ordered within each file so that no function calls a function defined before it.

If you wish to change any of the run time support functions, we strongly suggest that you make a library or module of your changed functions and link it before searching the supplied library, rather than actually changing the supplied library.

The following is a list of the run time support functions (in alphabetical order):

abs(n) - Returns the absolute value of "n".

alloc(n) - Allocate dymamic storage. Also see free(p).

This function attempts to allocate "n" bytes of system memory for use by the program. "n" is an unsigned integer (ie. -1 = 65535). The address of the memory is returned if the allocation succeeded, otherwise zero (0) is returned.

atoi(s) - Returns the integer value of the string "s".

bound(v,l,u) - Perform a range check.

This function returns one (1) if "v" is greater than or equal to "l" AND less than or equal to "u", otherwise zero (0) is returned. "v", "l" and "u" are all signed integers.

calloc(n,size) - Allocate dynamic storage.

This function attempts to allocate sufficient system memory for "n" items each of "size" bytes. Both "n" and "size" are unsigned integers. The address of the memory is returned if the allocation succeeded, otherwise zero (0) is returned. See also cfree(p).

ccexit() - Exit program with no cleaning up.

This function immediately exits a C program without closing open files. It is equivalent to the function _exit() described in the C book by Kerninghan and Ritchie.

ccioin() - Initialize the i/o and support package.

This function is called during the setup of the environment of every Intersoft C program. This function should never be called from a C program. It performs four functions. First, it initializes the i/o variables. Secondly, it initializes the dynamic memory allocation package. Thirdly, it parses the command line used to invoke the C program, including re-direction of the standard input and output files. Finally, it opens the standard input, output and error files.

This function causes a considerable portion of the Intersoft run time support functions to be included with your program. It is possible to alter the run time environment of C programs by changing this function (eg. removing command line parsing, adding i/o re-direction of the standard error file, etc.).

If your are using C to generate code for another machine you should write your own ccioin() function and include it in your source. It will replace the one supplied by us when you link it with your program. This will let you use the run time environment of your micro for development of the software, then by including your ccioin() function you can reconfigure your program for its new environment. See also ccmain() and ccsetup().

ccmain() - Initial entry point of all C programs.

This function performs any initialization required before C code may be executed (eg. In the CP/M version this is where the stack pointer and the reset vectors are initialized) then jumps to the routine ccsetup(). See also ccsetup() and ccioin().

ccsetup() - Set up, execute and terminate a C program.

This function calls ccioin() to initialize the remainder of the C environment, then calls the main function of the C program "main(argc, argv)", then calls the exit function "exit(0)" to terminate the program. See also ccmain() and ccioin().

cfree(p) - Deallocate dynamic storage.

"p" is the address returned by calloc(n,size).

ckopen(name,mode) - Open a file OR ELSE!

This function will open the file "name" with mode "mode" and return a valid file pointer or else it will print the message "Can't open <name>" and terminate the program.

copybyte(from,to,number) - Copy bytes.

This function copies "number" bytes from the address "from" to the address "to". "number" is an unsigned integer.

error(s,t) - Print an error message and exit.

This function prints the two strings "s" and "t" to the standard error file, then calls the exit() function to terminate the program.

exit(n) - Terminate a C program, closing all files.

"n" is the program termination code. Zero (0) means normal termination.

fclose(fp) - Close a file.

This function attempts to close a file. It returns non-zero if the file is closed correctly. "fp" is the value returned by the fopen() function for the file in question.

feof(fp) - Check for end of file.

This function returns non-zero if the file "fp" has reached end of file, otherwise zero (0) is returned.

fgets(s,fp) - Read a line from a file.

This function reads a line from the file "fp" (to the first newline character) and puts it in the string "s". The address "s" is returned if the string was read correctly, otherwise EOF is returned.

findeos(s) - Find the end of a string.

This function returns the address of the string terminator for the string "s".

fopen(name,mode) - Open a file.

This function opens the file "name" in the mode "mode" and returns a valid file pointer upon success or zero (0) upon failure. "name" is the file name as a string in the appropriate format for your operating system. "mode" is a string indicating how the file is to be accessed. Valid modes are "r" or "R" for read, "w" or "W" for write and "a" or "A" for append. The maximum number of simultaneously open files is given in the Implementation manual.

fprintf(fp,control,arg1,arg2,...) - Formatted output.

This function is similar to the printf() function. fprintf() performs the same operations as the printf() function except that it can operate on any file open with write or append access. "fp" is the file pointer Please refer to the printf() function for a description of the other arguments.

fputs(s,fp) - Write a string to a file.

This function writes the string "s" to the file "fp" and returns the address "s" upon success or EOF upon failure.

free(p) - Deallocate dynamic storage.

This function deallocates storage allocated by the function "alloc". "p" is the address returned by "alloc".

fscanf(fp,control,arg1,arg2,...) - Formatted input.

This function is similar to the scanf() function. This function performs the same operations as the scanf() function except that it acts on any file open with read access. "fp" is the file pointer. Please refer to the scanf() function for descriptions of the other arguments.

getc(fp) - Get a text character from a file.

This function returns the next character from the text file "fp". A special newline character '\n' is returned upon end of line, and a special integer "EOF" is returned upon end of file or upon an i/o error.

getchar() - Get a text character from the standard input file.

This function is equivalent to "getc(stdin)".

getdec() - Get a decimal number from the standard input file.

This function is equivalent to "getnum(stdin)".

getnum(fp) - Get a decimal number from a text file.

This function skips whitespace, then reads a string of six or less non-whitespace characters and attempts to convert it to a decimal number. Conversion stops with the first non-digit. The number may be signed (either + or -). The converted integer is returned.

gets(s) - Get a string from the standard input file.

This function is equivalent to "fgets(s,stdin)".

initbyte(to,val,number) - Initialize an array of bytes.

This function assigns "val" to "number" bytes (8-bit items) from the address "to" onward.

initword(to,val,number) - Initialize an array of words.

This function assigns "val" to "number" words (16-bit items) from the address "to" onward.

isalpha(c) - Test for alphabetic characters.

This function returns one (1) if "c" is an alphabetic character, otherwise zero (0) is returned.

isdigit(c) - Test for numeric characters.

This function returns one (1) if "c" is a decimal digit, otherwise zero (0) is returned.

ishex(c) - Test for hexidecimal characters.

This function returns one (1) if "c" is a hexidecimal digit, otherwise zero (0) is returned.

islower(c) - Test for lower case alphabetic characters.

This function returns one (1) if "c" is a lower case alphabetic character, otherwise zero (0) is returned.

isoctal(c) - Test for octal digits.

This function returns one (1) if "c" is an octal digit, otherwise zero (0) is returned.

isspace(c) - Test for whitespace characters.

This function returns one (1) if "c" is a whitespace character, otherwise zero (0) is returned. Whitespace characters are space, tab and newline ('\n').

isupper(c) - Test for upper case alphabetic characters.

This function returns one (1) if "c" is an upper case alphabetic character, otherwise zero (0) is returned.

itoa(n,s) - Convert an integer to an ascii string.

This function converts the integer "n" into its ascii representation (with sign) and puts it in the string "s".

max(a,b) - Return the maximum of two signed integers.

meminit() - Initialize the dynamic memory package.

This function frees ALL allocated space and initializes the dynamic memory variables. This function should be used with GREAT caution since not only will it deallocate all space allocated by a C program it will also deallocate the file control blocks and file buffers of all open files.

memleft() - Return the size free memory.

This function returns the number of bytes in the largest free memory block maintained by the dynamic memory package. The returned value is an unsigned integer.

min(a,b) - Return the minimum of two signed integers.

printf(control,arg1,arg2,...) - Formatted output.

The "control" argument is the address of a string. The "control" string contains characters to be printed or data conversion specifications. Each time a conversion specification is found in the control string the next of the "arg1", "arg2" arguments is interpreted and written. The format of a conversion specification is %-0min.maxA.

max - This specifies the maximum length of the argument in characters. If there are more characters in the converted argument they will be chopped from the right. The default value of this field is 132. Setting the field to zero will also result in a maximum length of 132. 0 <= max <= 32767.

. - This separates "max" and "min". It is not required if "max" is not specified.

min - The minimum width of the converted argument in characters. The default value of this field is zero. If the converted argument is less than the minimum width then the field will be padded according to the "-" and "0" arguments.

- - Specifies that the converted argument will be left justified. The default is right justification.

0 - Specifies that zeroes will be used as pad characters. The default pad character is a space.

A - The conversion character, one of
S or s - indicates that the argument is a string.

C or c - indicates that the argument is a character.
D or d - indicates that the argument is a decimal integer.
U or u - indicates that the argument is an unsigned decimal integer.
X or x - indicates that the argument is a hex integer (The leading 0x is not printed).
O or o - indicates that the argument is an octal integer (The leading 0 is not printed).

All but the '%' and the conversion character are optional in the conversion specification.

Note that if no conversion character is found that everything in the control argument from the current '%' character to the next '%' character will be output with no special formatting.

In the following examples we have bracketed the formatted output with colons (:) to indicate where the output begins and ends.

Example 1

```
printf("hi > %30.10s", "hello world !\n");

prints
:hi >                          hello worl:
```

Example 2

```
printf("hi > %30.10s\n", "hello, world !");

prints
:hi >                          hello worl
:
```

The newline is written since it was in the control argument, not in the truncated portion of the string argument.

Example 3

```
int a;
a = 123;
printf("+%u", a);

prints
:+123:
```

Example 4

```
int a;
a = -1;
printf("%010d", a);

prints
:-000000001:
```

Example 5

```
char *s;
s = "hello";
printf("%-10s",s);

prints
:hello     :
```

Example 6

```
printf("%010s", "hi");

prints
:00000000hi:
```

putc(c,fp) - Write a character to a text file.

This function writes the character "c" to the file "fp" and returns "c" if the write was successful, otherwise "EOF" is returned.

putchar(c) - Write a character to the standard output file.

This function is equivalent to "putc(c,stdout)".

putdec(n) - Write an integer to the standard output file.

This function is equivalent to "putnum(n,stdout)".

putnum(n,fp) - Write an integer to a text file.

This function converts "n" to its ascii form (possibly signed) and writes it to the file "fp". The last digit of the number is returned upon success, otherwise "EOF" is returned.

puts(s) - Write a string to the standard output file.

This function is equivalent to "fputs(s,stdout)".

scanf(control,arg1,arg2,...) - Formatted input.

The "control" argument is the address of a string. The control string contains whitespace, characters which must match the input file and conversion specifications which indicate how to convert data from the input file and store it in the "arg1", arg2" arguments. ALL of the "arg1", "arg2" arguments must be pointers to data, not the data items themselves. Each time a conversion specification is encountered it results in the next of the "arg1", "arg2" arguments being assigned.

Whitespace characters are the space, tab and newline ('\n') characters. All whitespace characters in the control argument are ignored.

When a non-whitespace character (not part of a conversion specification) is encountered in the control string it is expected to match the next non-whitespace character from the input file. Failure to match will cause scanf() to fail at that point and return the number of arguments successfully matched.

Scanf() returns the number of arguments successfully matched. It will return upon exhausting the control string, upon failing to match an argument, or upon end of file. If end of file is encountered scanf() returns EOF.

The format of a conversion specification is %*maxA where:

* - An optional flag indicating that assignment is to be suppressed. If this flag is specified the data is read from the input file but not assigned to one of the "arg1", "arg2" arguments.

max - The maximum field width. If this is specified the input file will be read until the end of the maximum field width or the next whitespace character.

A - The conversion character, one of:
- D or d - A decimal integer.
- O or o - An octal integer.
- X or x - A hexidecimal integer.
- H or h - A short integer (assigns to a character variable).
- C or c - An ascii character.
- S or s - An ascii string. The corresponding argument should be large enough to contain the string with a terminating '\0' (NULL).

One of the above must be present.

Example 1

```
int i;
scanf("%d", &i);
and the input line
        1000
will assign 1000 to i.
```

Example 2

```
int i, j, k;
scanf("%2d%2d%2d", &i, &j, &k);
and the input line
010101
will assign 1 to i, j, and k.
```

Example 3

```
int i, j;
scanf("%2d%*2d%2d", &i, &j);
and the input line
010203
will assign 1 to i and 3 to j.
```

Example 4

```
char a[30], b[30], c[30];
scanf("%s%s%s", a, b, c);  /* No mistake, the name of an array */
                           /* refers to the address of the array */
and the input line
      fee fie foe!
will assign "fee" to a, "fie" to b and "foe!" to c.
```

sprintf(ptr,control,arg1,arg2,...) - Formatted output.

This function is similar to the printf() function. sprintf() performs the same operations as the printf() function but operates on a string instead of a file. "ptr" is the address of the string into which the output is written. sprintf() assumes that the string is large enough to hold the output.

sscanf(ptr,control,arg1,arg2,...) - Formatted input.

This function is similar to the scanf() function. sscanf() performs the same operations as the scanf() function but operates on a string instead of a file. "ptr" is the address of the string from which the input is read.

strcat(s,t) - Concatenate string "t" to the end of string "s".

No check is made to verify that string "s" has enough storage allocated to hold string "t" as well.

strcmp(s,t) - Compare strings.

This function compares strings character by character. Zero is returned if the strings are identical. Otherwise the difference between the first pair of differing characters is returned (ie. s[i] - t[i] for the first i such that s[i] != t[i]). Therefore if this function returns a negative number then the string "s" is less than the string "t". If this function returns a positive number then the string "s" is greater than the string "t".

strcpy(s,t) - Copy string "t" to string "s".

No check is made to verify that string "s" is large enough to receive string "t".

string(n,c) - Create and initialize a vector of bytes.

This function allocates a vector of "n" bytes and initializes all elements of the vector to "c". "n" is an unsigned integer.

strlen(s) - Return the length of string "s".

system(s) - Execute a system command.

This function is not supported, it causes an error message to be displayed and execution to be terminated. "Attempted system call to <s>"

tolower(c) - Returns the lower case value of "c".

toupper(c) - Returns the upper case value of "c".

umax(a,b) - Returns the maximum of two unsigned integers.

umin(a,b) - Returns the minimum of two unsigned integers.

ungetc(c,fp) - Unread a character from a text file.

This function returns the character "c" to the file corresponding to the file pointer "fp" so that "c" will be the next character read from the file. Only one "unread" character may be outstanding at any time for a given file. The character "c" is returned upon success, otherwise EOF is returned.

TERSE REFERENCE GUIDE

The following description is not intended to be rigorous. It provides a terse general guide to the syntax of Intersoft C.

Meta-symbols are enclosed in angle brackets, the "::=" operator is used for "is defined as", possible values are listed on separate lines, NULL means that it is legitimate to put nothing in that place. Whitespace is allowed wherever a space has been left between meta-symbols and/or terminal symbols. Square brackets "[]" have no significance other than as terminal symbols.

```
<data.or.func> ::=
     <data.def>
     <func.def>


<data.def> ::=
     <char.def>
     <int.def>

<char.def> ::=
     char <identifier.list> ;

<int.def> ::=
     int <identifier.list> ;


<identifier.list> ::=
     NULL
     <identifier.list.2>

<identifier.list.2> ::=
     <ident.def>
     <ident.def> , <identifier.list.2>

<ident.def> ::=
     <identifier>
     *<identifier>
     <identifier> [ <constant.expr> ]

<identifier> ::=
     <alpha>
     <alpha><alphanum.list>

<alpha> ::=
     <"a" to "z", "A" to "Z" and underscore>

<alphanum> ::=
     <All "<alpha>" characters and "0" to "9">
```

```
<func.def> ::=
     <identifier> ( <name.list> ) <stmt>
     <identifier> ( <name.list> ) { <var.def> <stmt.list> }

<name.list> ::=
     NULL
     <name.list.2>

<name.list.2> ::=
     <identifier>
     <identifier> , <name.list.2>

<var.def> ::=
     NULL
     <char.def> <var.def>
     <int.def> <var.def>

<stmt> ::=
     { <stmt.list> }
     if ( <expr> ) <stmt>
     if ( <expr> ) <stmt> else <stmt>
     while ( <expr> ) <stmt>
     do <stmt> while ( <expr> ) ;
     for ( <expr> ; <expr> ; <expr> ) <stmt>
     switch ( <expr> ) <stmt>
     return <expr> ;
     ;
     <expr> ;
     break ;                          -- Valid in loop or switch!
     continue ;                       -- Valid only in loop!
     case <constant.expr> : <stmt>    -- Valid only in switch!
     default : <stmt>                 -- Valid only in switch!

<stmt.list> ::=
     NULL
     <stmt> <stmt.list>

<expr> ::=
     <primary>
     * <expr>
     & <expr>
     - <expr>
     ! <expr>
     ~ <expr>
     ++<lvalue>
     --<lvalue>
     <lvalue>++
     <lvalue>--
     <expr> <binaryop> <expr>
     <expr> ? <expr> : <expr>
     <lvalue> <asgn.op> <expr>
```

```
<primary> ::=
     <identifier>
     <constant>
     ( <expr> )
     <primary> ( <expr.list> )
     <ptr.expr> [ <expr> ]

<ptr.expr> ::=
     <A pointer expression or array name, see <expr>>

<lvalue> ::=
     <identifier>
     <ptr.expr> [ <expr> ]
     * <expr>
     ( <lvalue> )

<constant.expr> ::=  -- An expression involving only constants.

<constant> ::=
     <number.const>
     <character.const>
     <string.const>

<number.const> ::=  -- Values are all modulo 65536
     <decimal>      -- Format is iddddd (i > 0) (0 <= d <= 9)
     <octal>        -- Format is 0ooooo (0 <= o <= 7)
     <hex>          -- Format is 0xhhhh (0 <= h <= F)

<character.const> ::=
     '<char.or.escapedchar>'

<string.const> ::=
     "<char.or.escapedchar.list>"

<char.or.escapedchar.list> ::=
     NULL
     <char.or.escapedchar> <char.or.escapedchar.list>

<char.or.escapedchar> ::=
     <Printable ascii and all escape characters
      defined in the section "How to program in C">

<expr.list> ::= NULL
     <expr.list.2>

<expr.list.2> ::=
     <expr>
     <expr> , <expr.list.2>

<binaryop> ::=
     *
     /
     %
     +
     -
```

```
      >>
      <<
      <
      >
      <=
      >=
      ==
      !=
      &
      ^
      |
      &&
      ||
      ,

<asgnop> ::=
      =
      +=
      -=
      *=
      /=
      %=
      >>=
      <<=
      &=
      ^=
      |=

<preprocessor> ::=
      #define <identifier> <token-string>
      #include <<filename>>
      #include "<filename>"
      #ifdef <identifier>
      #ifndef <identifier>
      #else
      #endif
      #asm
      #endasm
      #line <token-string>  -- Note: No effect on program!
```

COMPILER DIAGNOSTICS

Constant required, 1 used

Only constants are allowed.

Constant required, zero used

Only constants are allowed.

DEFAULT valid only in SWITCH

A "default" statement was found outside a "switch" structure. Check the function's block structures.

Global symbol table overflow

The compiler ran out of space in the global symbol table. See the section titled "How to Use the Compiler" to find out how to define a larger global symbol table.

Hex digit > F forced to F

All numeric constants preceeded by "0x" are assumed to be hexidecimal (base 16), the digits are 0-9 and a-f.

Illegal address with unary &

The unary "&" operator may be used only to get the address of an item for which storage has been allocated. Example: "&intval" is legal, "&0xff" is not.

Illegal argument name

Only simple symbol names may be used as argument names.

Illegal function name or declaration

Only simple symbol names may be used as function names.

Illegal symbol name ignored

Usually caused by non-alphanumeric characters within a symbol name.

Insufficient memory!

Your machine does not have enough memory for the compiler to execute. Caused by overestimating the requirements for one of the compiler options or by trying to compile large programs. Split the program into smaller compilation modules.

Invalid expression, evaluated as FALSE

Caused by a syntax error in an expression.

Literal pool overflow

The compiler has insufficient space to store the string and character constants. Please refer to the section titled "How to Use the Compiler" to find how to expand the literal pool. The literal pool is written at the end of each function so estimate the required literal pool size by the requirements of the function with the most character and string constants.

Local symbol table overflow

See "Global symbol table overflow". This message refers to the symbol table which keeps track of variables which are local to a function.

Lvalue not found

The indicated item is not a unit of storage and therefore cannot have anything assigned to it. Note: "0xFF = 3" is illegal but "*0xFF = 3" is legal (it stores 3 into memory location 0xFF).

Missing WHILE in DO...WHILE

Either you forgot it or the block structure of the function is corrupted.

Missing apostrophe in character constant inserted

A simple syyntax error, usually caused by not escaping an apostrophe in a character constant.

Missing apostrophe, part of line not preprocessed

A simple syyntax error, usually caused by not escaping an apostrophe in a character constant.

Missing argument name

The function has been called previously with more arguments than are present in the function definition.

Missing bracket

The language requires an opening bracket at the indicated column.

Missing colon

The language requires a colon at the indicated column.

Missing comma in argument list

Arguments must be separated by commas.

Missing open parenthesis

The language requires an opening parenthesis at the indicated column.

Missing quote inserted at end of line

If you wish to extend a string beyond one line you must use the escape character as the last character in the line.

No loops in effect

The indicated construct is valid only within a loop structure.

No loops or switches in effect

The indicated construct is valid only within a loop or switch structure.

Octal digit > 7 forced to 7

All numeric constants beginning with zero are assumed to be in octal (base 8).

Preprocess buffer overflow, line truncated

Either the source line is very long or it contains macros which have long definitions. Please refer to the section titled "How to Use the Compiler" to find out how to specify that a longer source line be allowed.

Semicolon missing

The language requires a semicolon at the indicated column. It is also possible that previous errors in the source statement have overcome the compiler's error recovery abilities. If this is the case the compiler will produce these messages until the end of the statement is encountered.

Symbol already defined

The symbol has been defined previously.

Too many nested includes

Too many nested loops and switches

Unknown preprocessor command ignored

Unrecognized option

Value too large, 0377 used

An escaped constant within a character or string constant is too large.

Write error

A write error has occurred on the output file. The most common cause is insufficient disk space.

Wrong type or number of arguments

Function argument definitions do not match the argument list in the function declaration.

CHANGES FROM INTERSOFT C V2.0

The following features have been added to the language. Descriptions of how to use them may be found in the section of this manual containing the reference guide.

1) Character escapes (\n, etc.).
2) The assignment operators (+=, -=, etc.).
3) The logical AND and OR operators (&&, ||).
4) The conditional operator (?:).
5) The comma operator (,).
6) Conditional compilation (#ifdef, #ifndef, #else and #endif).
7) All keywords are case insensitive.
8) Constant expressions are allowed in array definitions and in the "case" statement.

The compiler accepts special escape sequences for the characters {, }, [,], |, ~ and ^. See the section titled "Introduction".

The command interface to the compiler has been redesigned. The new command interface contains many new options. Please refer to the section titled "How to Use the Compiler".

Several new functions have been added to the run time support library. alloc(), free(), meminit(), memleft(), calloc(), cfree(), feof(), ungetc(), fprintf(), sprintf(), scanf(), sscanf() and fscanf(). Please refer to the section titled "Run time support" for descriptions of these new functions.

The compiler diagnostics have been improved considerably. Please refer to the section titled "Compiler Diagnostics".

The compiler now generates better code.

Limited constant folding has been added.

A problem with using the "continue" statement within a "switch" statement has been corrected.

The compiler now correctly scales values added to integer pointers. "iptr+3" would add three to the integer pointer "iptr" in version 2.0. Now this expression adds six so that "iptr" is adjusted to point to the third integer further on in memory.